Atelier B

Atelier B Translators

User Manual

version 4.6

ATELIER B
Atelier B Translators User Manual
version 4.6

Document made by CLEARSY.

This document is the property of CLEARSY and shall not be copied, duplicated or distributed, partially or totally, without prior written consent.

All products names are trademarks of their respective authors.

CLEARSY
Maintenance ATELIER B
Parc de la Duranne
320 avenue Archimède
Les Pléiades III - Bât. A
13857 Aix-en-Provence Cedex 3
France

Tel 33 (0)4 42 37 12 70
Fax 33 (0)4 42 37 12 71

mail : contact@atelierb.eu

Contents

1 Description of this manual **1**

1.1 Aim . . . 1
1.2 Required Prior Knowledge . . . 1
1.3 Overview of this Manuel . . . 2
1.4 How to use this Manual . . . 2
1.5 Conventions and syntax . . . 2
1.6 Related Documents . . . 3

2 Presentation of the Software **5**

2.1 Purpose . . . 5
2.2 Recommended Environment . . . 5
2.3 Provided Services . . . 6
2.3.1 Preface: Reasons for two-pass translation . . . 6
2.3.2 Automatic translation service of a B0 implementation into target language . . . 8
2.3.3 Linking service . . . 8
2.3.4 Basic Machines . . . 9

3 Use principles **11**

3.1 Operating Modes . . . 11
3.1.1 Using the Translator via the GUI* . . . 11
3.1.2 Using the translator in batch* mode . . . 12
3.1.3 Using the Translator from the command line . . . 13
3.1.4 Compiling and Executing the Code Produced . . . 14
3.2 Inputs and Outputs . . . 15
3.2.1 Messages Generated by the Ada Translator . . . 15
3.2.2 Using the translator from the command line or the Batch mode* . . . 15
3.2.3 Using the translator from Atelier B's GUI* . . . 15
3.2.4 Files . . . 15
3.3 Precautions for Use . . . 16

3.3.1 Important Proof Related Warning . . . 16
3.3.2 Important Warning Relating to the Values of the MAXINT and MININT Constants . . . 16
3.3.3 Size of Lines Produced . . . 16
3.3.4 Compatibility of the Translator with Atelier B . . . 17
3.3.5 Naming Modules and Projects for the Ada and HIA Translators . . . 17

4 Operating scenarios **19**
4.1 Developing a native B project . . . 19
4.1.1 Principle . . . 19
4.1.2 Informal Example Specifications . . . 19
4.1.3 Project Architecture and B Code . . . 20
4.1.4 Integrating Components into Atelier B . . . 23
4.1.5 Unitary Translation of the Produced Code . . . 23
4.1.6 Project linking . . . 26
4.1.7 Compiling and Executing the target code . . . 26
4.2 Developing a Heterogeneous B/Target Language Project . . . 26
4.3 Developement of a Heterogeneous B/HIA project . . . 29

5 Complete List of Services **31**
5.1 Unitary Translation of an Implementation . . . 31
5.2 Project Linking . . . 31
5.3 Translating a Project with Traces . . . 31
5.4 Restriction of translators use . . . 33

6 Glossary **35**

A Basic Machines **37**
A.1 Principle . . . 37
A.1.1 Definition . . . 37
A.1.2 Usefulness . . . 37
A.2 Description of the Basic Machines Delivered with Atelier B . . . 37
A.3 Writing a Basic Machine . . . 37
A.3.1 The B Specification writing method . . . 38
A.4 Target code (interface and body) writing method . . . 38
A.4.1 Target Code writing method for Ada . . . 39
A.4.2 Target code writing method for HIA . . . 40
A.4.3 Target code writing method for C++ . . . 40
A.4.4 Target code writing method for C . . . 41
A.4.5 The “B Link File” . . . 42

B B0 specifities accepted by the HIA translator 43
B.1 Introduction 43
B.2 Arrays translation 43
B.2.1 Principle 43
B.2.2 Example 43
B.3 Record translation 44
B.3.1 Principle 44
B.3.2 Example 44
B.4 Formal parameters 45

List of Figures

2.1 Files produced for the `mach` interface 8
2.2 Files produced for the `mach` body 8

3.1 Translation of an implementation in batch mode 12
3.2 Translation of an implementation using the command line 14

4.1 Specification and implementation of the stack component 21
4.2 Specification and implementation of the display_stack component 22
4.3 Specification and implementation of the interface_stack component 24
4.4 Specification and implementation of the demo component 25
4.5 Project dependency graph 26
4.6 Compilation of an Ada code produced by the Ada translator 27
4.7 Executing the Ada program 28
4.8 Correspondence between the physical instances of the project and their access path in Ada, C and C++ 28

5.1 Translation of an implementation from the command line 31
5.2 Options available when performing the unitary translation of a component 32
5.3 Options available during linking 32

A.1 Basic machines supplied with Atelier B 38
A.2 Translation of the B0 types in Ada, HIA, C and C++ 39
A.3 Example of a “B Link File” 42

Chapter 1

Description of this manual

1.1 Aim

This user manual applies to the following softwares :

- Ada Translator, since version 4.6[1]
- HIA Translator, since version 4.6[2].
- C/C++ Translator, since version 4.2[3]

From now on, when we will refer to the Translator software, this will mean that we are evoking the Ada, HIA, C or C++ Translators, indifferently. When we desire to differentiate between the software, we will clearly write Ada Translator, HIA Translator, C Translator or C++ Translator. In the same way, when we will refer to the target language, that will mean that we are referring to Ada, HIA, C or C++ Language.

The aim of this user manual is to make the required knowledge available to the persons using the Translator. It has a double goal:

- to enable these persons to learn progressively.
- to serve as a reference to identify the behavior of this software.

To do this, the required prior knowledge, how to access the manual according to the user's requirements, notation conventions used and useful reading will be exposed.

1.2 Required Prior Knowledge

The reader of this manual is assumed to be trained to B and to the target language, as well as in the use of Atelier B and the target compiler.

[1]The ADA translator translates all B0 implementations into ADA code in conformance with the Ada-83[ADA-83] and Ada-95[ADA-95] standards.

[2]The HIA translator translates into "High Integrity Ada" code, syntactically in conformance with the SPARK norm described in [SPARK]. In return for some restrictions on the B language in input, it generates a more simple code, very close to the B0 and "safer"

[3]This translator tranlates all the implementations into ANSI-C[ANSI-C] or ANSI-C++[ANSI-C++] code.

1.3 Overview of this Manuel

Chapter 2 presents the Translator's aims. The supported environments and the general translation paradigm are also described.

Chapter 3 details the operation principles of the Translator. Its use within Atelier B is described, along with its use through the command line of a shell* . The chapter ends with a description of the precautions to be taken when using a translator.

Chapter 4 uses a simple example to illustrate the full development cycle for a B project translated into the target language. The case of a B project whose translation into the target language is merged into a larger project written originally in the target language is touched on.

Chapter 5 sums up the program use options. It then details the use of the Translator which can be used to manage project configurations by using the SCCS* tool.

Finally, chapter 6 explains the technical terms used in this document.

Appendix A explains the procedure to follow when developing basic machines.

1.4 How to use this Manual

A novice user of the Translator may read only Chapters 2, 3 and 4, as an initial introduction. The example presented in this last chapter provides a complete illustration of the use of the program and should allow the user to come to terms with the Translator progressively and completely.

Once he gets acquainted with the program, an experienced user will find in Chapter 5 a summary of the options available when using the translator.

1.5 Conventions and syntax

- The "computer objects" such as file names, window names or menu items are written using a non-proportional font as shown in the example below:

  Machine `MM.mch`.

  The input/output exchanges between the user and the program are described using the same font. To differentiate inputs from outputs, the messages generated by the program will be preceded by the > sign, as shown in the example below:

  ```
  ls
  > AA.mch    AA_1.imp  SCCS
  ```

- Words that are explained in the Glossary *(chapter 6, page 35)* are followed by an asterisk, as shown in the example below: "The GUI* user".

- The paragraphs describing specificities of one or several translators are preceeded with the symbol ↻ and written in a specific font, as follows:

  ↻ This section does not concern the translator...

1.6 Related Documents

The bibliography *(page 47)* provides a list of documents that allow a novice user to learn how to use Atelier B and the target language, and which serve as a reference base for experienced users.

Chapter 2

Presentation of the Software

2.1 Purpose

The purpose of the Translator software is to perform an automatic translation of the B0 implementations of a project into target source code. The target source code may be compiled in order to realise an independent project, or integrated into a native target language development.[1].

The translators are able to translate into Ada, HIA, C or C++ the whole B0 language. There are no restrictions, especially as regards the naming of identifiers : any identifier which may enter into conflict with the target language is renamed by the Translator.

Important remark: the HIA translator works with a B0 language that has some specificities detailed in appendix B.

So, any component that is successfully analysed by the B0 checker can be translated into the target language. In the rest of this manual, any implementation of a component for which the B0 checker is successfully executed will be called a B0 implementation.

2.2 Recommended Environment

The Translator is intended for execution on the same platforms as the Atelier B. The Translator generates a portable target code, in conformity with the current norms.

Program options are used to configure the generated code in order to match the target system and the target compiler [2].

In its 4.6 version, the code produced by this program has been tested with a GNU compiler in the following environments:

- Sun workstation running Solaris 2.6, 7 or 8
- Hewlett-Packard workstation running HP-UX 10.20
- PC type computer running Linux 2.2

[1] This function is used to integrate, into a project written in the target language, the translation of the secure components written in B.

[2] In this way, a cross-translation (similar to cross-compiling) can be performed .

Important Remark : The user must have a development environment for the complete target language, since no compiler or language interpretation tools are supplied with the Translator.

2.3 Provided Services

2.3.1 Preface: Reasons for two-pass translation

This section presents technical aspects that may initially be ignored by a novice user *(who may in this case refer directly to paragraph 2.3.2)*.

The Translator is used to translate any B0 implementation into target language automatically. A B0 implementation is not translated into a source code in target language in a single call (or pass) of the translator.

Two passes are used :

- Unitary translation: during this first pass, each B0 implementation is translated into an "object" file, independently from the other implementations. These "object" files produced during the first pass can be re-used by several projects, allowing the creation of B libraries .
- linking : during this second pass, the Translator produces in particular the target language files *(refer to paragraph 2.3.3)*. These target language files are specific to the current project and cannot be re-used in another project.

The motivations for this two-pass translation mechanism *(production of object files then linking)* are linked to the following aims of the Translator :

- Complete translation of the B0 language.
- Accurate and efficient translation.

Complete translation of the entire B0 language

A certain number of identifier clashes can occur when translating from B0 into the target language :

- The B0 language distinguishes between `ident` and `IDENT`, which is not the case for the Ada language.
- The `void` identifier in B0 clashes with the `void` keyword reserved in C++.
- The B0 language allows you to write identifiers that comprise more than one '_' successive character *(such as for example* `ident__ifier`*)*. These identifiers are not valid Ada identifiers.
- Some valid B0 identifiers clash with the target language reserved words. For example, `package` is a valid B0 identifier.

Therefore it is clear that a phase of identifier clash resolution, covering an entire project is necessary to perform an automatic and systematic translation from the B0 language to the target language.

Reliable and high performance translation

↺ This section does not concern the HIA translator which manages arrays and records translation differently. Refer to the B appendix for more details.

Reliable translation of B0 arrays requires determining by inference the types of the arrays used and automatically generating the related "array types". This action must be efficient because two arrays that have the same inferred type in B0 must be translated by two arrays that have the same generated type in the target language, so as to be able for example to copy and compare these arrays.

The same problem occurs during the translation of B0 records : it is necessary to infer a declaration of the associated record type.

A generation and resolution phase for array and records types that apply to the entire project is therefore necessary for a reliable and high performance translation of the elements of this kind in the B0 language *(which does not explicitly type arrays and records)*, into the target language : C, C++, or ADA *(which imposes a strict explicit typing of arrays and records)*.

Resolving implicitly glued data and renaming

Resolving implicitly glued data and renaming in a strictly typed language such as Ada or C++ can only be carried out during a linking phase when the Translator has an overall view of the entire project.

Creation of the component instances

↺ This section applies only to the HIA translator

The linker is responsible for the copy *(physical copy of the files)* of the files for each instance of a project component.

So if in a project two instances `M1` and `i1.M1` of a component `M1` are used, then the linker creates two packages:

- The package `M1`, in `m1.ads` and `m1.adb`
- The package `i1_M1`, in `i1_m1.ads` and `i1_m1.adb`. This package is obtained by copying the package `M1` and replacing all the occurences of `M1` by `i1_M1`.

Valuation of formal parameters

↺ This section applies only to the HIA translator.

The HIA translator declares the formal parameters in the associated packages. It then has to define in each package having formal parameters, not only the name and the type of those parameters, but their value too.

Therefore, the effective value of a formal parameter is known only during the module importation by another project module *(example : in* `M1`*, the clause* `IMPORTS M2(10)` *: it is in* `M1` *that we know the value* `10` *of the formal parameter* `param` *of* `M2`*. But it is in* `M2` *that* `param : constant INTEGER := 10` *must be written.*

Target language	After unitary translation	After linking
Ada	`mach.str`	`mach.ads`
C	`mach.spe`	`mach.h`
C++	`mach.spe`	`mach.h`

Figure 2.1: Files produced for the `mach` interface

Target language	After unitary translation	After linking
Ada	`mach.bod`	`mach.adb`
C	`mach.bdy`	`mach.c`
C++	`mach.bdy`	`mach.cpp`

Figure 2.2: Files produced for the `mach` body

The linker has then to set up the effective values of the formal parameters, corresponding to the components use in the B project to translate.

2.3.2 Automatic translation service of a B0 implementation into target language

This service takes as its input a BO implementation `mach.imp` and produces the following files :

- Two files for the `mach` interface. One after unitary translating and one after linking. The file names are given in table 2.1.
- Two files for the `mach` body. One after unitary translating and one after linking. The file names are given in table 2.2.
- An object file which describes the symbols imported and exported by `mach`. By default, this file is named `mach.blf`. Among other things, this file is used to produce the list of substitutions to be applied to the object interface and body files so as to produce the final target files.

2.3.3 Linking service

Case of an Autonomous Project

In the case of an autonomous project, the linking service takes as its input the name of the specification of an implementation destined to become the entry point for a project as well as the name of the start-up module to be created, and it creates :

- All the source files *(interfaces and bodies)* for the project components.

  ↻ For the HIA translator, files are duplicated for each instance of a component.

- The target code for the start-up module of the project.

- The interface and the body of the `sets` module (in Ada) or `SETS` (in C/C++) for the inferred array and record types, the concrete constants, the abstract sets and the predefined elements (`succ`, `pred`, `MAXINT`, `MININT`, ...).

  ↻ FOR THE HIA TRANSLATOR, THIS FILE CONTAINS ONLY THE PREDEFINED SETS AND FUNCTIONS.

- The `makefile` file which will be used to generate the project.

To be usable as the entry point of a project, an implementation must have one, and only one operation, with no input or output parameters. However, the name of this operation can be freely defined.

The link editor recursively scans the import links for this machine and thereby translates into target "terminal" code all the object files used by project. Object files are sought for in the `lang` directory of the project during the translation, then in the `lang` directory of used libraries, in the order they were declared.

Case of a heterogeneous project

In the case of a heterogeneous project, the link editor takes as an input the name of the specification of an implementation destined to become an entry point for a project, as well as the name of a start-up module to be created, and produces :

- All the source files (interface and body files) for the project components.
- The target code for the access point to the B project. This module enables the initialization of all the B components, and then access to all the data and the operations of these machines.
- The interface and the body of the `sets` module (in Ada) or `SETS` (in C/C++) for the inferred arrays types, the concrete constants, the abstract sets and the predefined elements (`succ`, `pred`, `MAXINT`, `MININT`, ...).
- A `makefile` skeleton that allows the project to be generated.

The link editor recursively scans the import links for this machine and thereby translates into target "terminal" code all the object files used by project[3]. Object files are sought for in the `lang` directory of the project during the translation, then in the `lang` directory of used libraries, in the order they were declared.

Important note : Libraries must be translated before the project that uses them! If this is not the case, project linking will fail as the object files of the library do not exist.

2.3.4 Basic Machines

The Translator is supplied with B specifications and the object files for a set of basic machines that facilitate the setting up of formatted inputs and outputs, tables indexed by enumerated sets or by intervals, ...

These machines are described in Appendix A.

[3]On the other hand, object files from unused components in the project are not translated.

Chapter 3

Use principles

3.1 Operating Modes

The Translator may operate in the following three modes:

- In the context of a work session using Atelier B GUI* .
- In the context of Atelier B batch mode.
- Through the command line

3.1.1 Using the Translator via the GUI*

Translating a BO implementation

To translate an implementation `implementation.imp` of a project `proj` using the GUI of Atelier B, the following operations must be carried out[1] :

- Using the mouse, select the implementation `implementation.imp`.
- Click on the `Translator` button. From the window that is displayed, select the target language with `Language` and then select `Selected Only` for `Components`. Then click on the `OK` button. The translation process is then started.

Remark: As described in sub-section 2.1, a component must have successfully passed the B0 checking step before it can be translated.

Therefore, if this step was never performed or if the component was modified since the last check was performed, the B0 checking tool will first be invoked for the implementation to translate.

This tool may induce the Type Checker invocation for this implementation. If necessary, all required machines may also be analyzed by this tool.

Sub-section 3.1.2 provides an example of the messages produced by the Ada translator during translation.

[1]Project creation and management operations are not described in this manual, the reader will find their description in [ATB1].

Translator	Command
Ada	`adatrans implementation_1`
HIA	`hiatrans implementation_1`
C	`ctrans implementation_1`
C++	`c++trans implementation_1`

Figure 3.1: Translation of an implementation in batch mode

In the same way, it is possible to translate several implementations during a single operation by selecting them all.

Global project linking

To perform linking on all a project's components, use the mouse to select the file that serves as the project's entry point. Then click on the `Translator` button and choose the target language from the `Language` menu, and `All` for `Components` options. The name of the project start-up module is the name of the B project.

Sub-section 3.1.2 provides an example of the messages produced by the Ada translator during the translation.

3.1.2 Using the translator in batch* mode

Translating a B0 implementation

To translate an implementation `implementation_1.imp` of project `proj` using Atelier B batch* mode, type the command given by the 3.1 table.[2]:

The translator [3] will then produce the following type of output[4] :

```
 Translating into ADA the file implementation_1
> Entering B0->Ada mode ...
> Creating B Extended Tree
> Creating package specifications (/home/B/projet/lang/implementation.str)
> Creating package body (/home/B/projet/lang/implementation.bod)
> Creating B Link file (/home/B/projet/lang/implementation.blf)
> Free B Extended Tree
>
>
> Translation into ADA successful
```

If the command is repeated without changing the B0 source file, a message similar to the following is obtained :

```
 Component implementation_1 is already translated
```

[2]The project creation and management operations are not described in this manual, the reader will find a full description in [ATB1].

[3]The example output given is produced by the Ada translator, the output is similar for the other translators.

[4]The example given is produced by the Ada translator in verbose mode. By default, a more concise output is obtained.

It is then possible to force the translation of the component by disabling the dependence checks function *(command* `ddm` *or* `disable_dependence_mode`*)*, and then calling the translator. Conversely, the dependence check function can be re-enabled by typing `edm` or `enable_dependence_mode`.

Global project linking

To perform the project linking, first determine the project entry point *(in this case, it is* `input_1.imp`*)* and type the command:

Target language	Command
Ada	`ada_all entry_1`
HIA	`hia_all entry_1`
C	`ccompile entry_1`
C++	`c++all entry_1`

The translator [5] produces an output similar to the following[6] :

```
> Entering project mode
> Calling B linker
> Entering project mode
> Analysing module entree
> (entry) exports (constants)
> Analysing machine constants
> Creating makefile
> Creating ada source code for executable module (project.bod)
> Analysing instance this
>       This instance does not have a SEES clause
> Analysing instance this.ref_constantes
>       This instance does not have a SEES clause
> Analysing imported variables of instance this
>       This module does not import any variable
> Analysing imported variables of instance this.ref_constantes
>       This module does not import any variable
> Creating template for package sets
> Installing project
> Creating temporary bed file /tmp/blka04747
> Executing "/home/ada/bed/bed -s /tmp/blka04747 -i /home/B/project/lang/makefile.blf
> -o /home/B/project/lang/makefile"
...
 > Executing "rm /tmp/blka04747"
> Freeing allocated objects
>
> ADA translation successful
```

3.1.3 Using the Translator from the command line

Chapter 5 presents the provided command line options to use when using the Translator. Note that this possibility should be reserved to experienced users since :

[5] The sample output given here is produced by the Ada translator. The other translators produce similar output

[6] The example given is produced by the Ada translator in verbose mode. By default, a more concise output is obtained.

Translator	Software
Ada	tbada
HIA	tbhia
C++	tbcpp
C	tbcpp -c

Figure 3.2: Translation of an implementation using the command line

- When used from the command line, the translator requires you to provide many parameters : project database directory (`bdp`), project translator directory (`lang`), full path for the linker replacement tool (`bed` tool), path to the directory holding the information of the `LIBRARY` project, ...
- If you use the Translator from the command line, you must deal yourself with the dependences. For instance, if B source files are modified, you must not forget to re-translate each component that uses them, ...

The array 3.2 gives the name of each translator.

In the following examples, we will assume that we want to translate into C++ a B project named `my_project`.

We must therefore know the following information :

Identifier	Meaning
`${My_Project}/lang`	Path to the translation directory
`${My_Project}/spec`	Path to the B source code
`${AB}/press/lib`	Path to the `LIBRARY` project provided with Atelier B
`Entry`	Project entry point

1. Case of an unitary translation : Translation of the implementation `Component_1.imp` of project `My_Project` :

```
tbcpp -i Component_1.imp -P ${my_project}/lang
      -I ${AB}/press/lib/spec -L ${AB}/press/lib/lang/cpp -w
```

2. Case of the translation of the project entry point with linking: translation of the entry point `Entry_1.imp` and linking.

```
tbcpp -o My_Project -e Entry -E bed -P ${My_Project}/lang
       -I ${AB}/press/lib/spec -L ${AB}/press/lib/lang/cpp  -w %$
```

This is only a little example, the reader should read chapter 5 to get the full list of services.

3.1.4 Compiling and Executing the Code Produced

- Execution on the machine that hosts Atelier B. From the shell* window, simply select the project's `lang` directory and type `make`.

- Execution on a target machine. Transfer, the full contents of the project's `lang` directory to this machine and type `make`. This directory contains all the files required for compiling *(including files from the translation of libraries.)*.

The `makefile` file also defines the goal `clean` used to delete all binary files produced by the target compiler. To perform this operation, type `make clean`.

3.2 Inputs and Outputs

3.2.1 Messages Generated by the Ada Translator

3.2.2 Using the translator from the command line or the Batch mode*

The Translator produces messages that describe its operation from the standard output *(i.e,* `stdout`*)*. It may also produce warning messages from this same standard output, as well as error messages from the error output *(i.e,* `stderr`*)*.

Therefore, a script that starts a complete translation scenario may retrieve the results obtained in `stdout` and any error messages from `stderr`, as shown in the following example [7]:

```
#!/bin/sh
# Script that translates all  the implementations of the current directory
rm -f /tmp/res
rm -f /tmp/err
for f in *.imp
do
    echo ‘‘Translation of $f’’
    tbada -i $f >> /tmp/res 2>> /tmp/err
done

echo ‘‘Translation results :’’
cat /tmp/err | less
echo ‘‘Translation errors :’’
cat /tmp/err | less
```

3.2.3 Using the translator from Atelier B's GUI*

The messages are integrated inside Atelier B's GUI* .

3.2.4 Files

The Translator reads and writes the files using the standard operating system API* . The points to watch are therefore:

- Ensuring that the user who starts the program has the following rights:
  - The right to read from the source and `bdp` directories of the project libraries used, and this for all files in these directories.

[7] The HIA translator is used in this example

  - The right to write in the project's `bdp` and `lang` directories.

- Ensure that enough space remains in the project file system.

If these instructions are not followed, the Translator generates an error message that explains the problem caused by the system. These messages are provided to the program by the system. It may therefore be necessary to configure the system or the user's account in order to modify the characteristics of these messages *(the language used by the system to provide the error messages may sometimes be chosen by environment variables or by other methods.)*.

3.3 Precautions for Use

3.3.1 Important Proof Related Warning

The target code generated by the Translator is only valid if the components that are translated are completely proven.

The Ada translator allows you to translate projects whose components are not completely proven in order to offer users greater flexibility in their development. However, the translation of a component that is not yet proven causes the generation of a warning message, and the generated code comprises a warning comment in its header.

Indeed, the generated code might not be compilable *(in the case of a design error in B that is only detected at the proof stage)*, or can raise exceptions during its execution *(e.g, in the case of access to an invalid index in a table)*.

3.3.2 Important Warning Relating to the Values of the MAXINT and MININT Constants

The ADA Translator allows you to redefine the values of `MAXINIT` and `MININT` in order to translate the code destined for a target system whose architecture differs from the one of the host system.

The use of a MAXINT and/or MININT value that is different from the one used by the prover will cause a result that cannot be guaranteed.

By default, the Translator is compatible with the prover. It is recommended to contact ClearSy for advice if you want to modify the values of these constants.

3.3.3 Size of Lines Produced

Some target compilers, only allow files with lines that do not exceed a certain size as inputs.

For this reason, the Translator checks the length of the lines that they produce. This length may be set by the user with the `-l` and `-t` options, whose default values are respectively `80` and `4`.

If the user modifies these values, they must ensure:

- That the maximum line size supplied does not exceed the capacity of the target compiler used.
- That the maximum line size supplied does not inhibit the code generation. Remember that the operation calls are prefixed by the names of the machines that define them. It is easy to see that the generation of long lines that cannot be broken is possible simply by giving "long" names to machines and operations. It is then necessary to ensure compatibility between this choice and the choice of the maximum length of the lines produced. The "Limits Checker" tool in Atelier B may assist the user in setting these limits.

3.3.4 Compatibility of the Translator with Atelier B

↻ This restriction does not concern the ADA and HIA translators.

It is always necessary to ensure that the version of the Translator is compatible with Atelier B tools.

3.3.5 Naming Modules and Projects for the Ada and HIA Translators

↻ This restriction concerns the ADA and HIA translators only.

The only naming conflicts that the linker cannot resolve are conflicts that occur in module names. The linker cannot rename the modules as the Ada language imposes that the files that represent a compiled unit *(i.e, a procedure or a package)* have the same name as this unit[8].

For example, if a module or project name is in conflict with an Ada keyword, the linker cannot resolve the conflict and the translation into Ada will fail.

The restrictions that apply to the module and project names are as follows:

- In a project, no module may have the same name as the project.
- The project name must not be in conflict with the target language.

It should be remembered that the Ada language does not differentiate between upper and lower case characters, therefore the rules above must be applied without reference to any possible use of lower case. Therefore a project `PROJ` cannot contain a `proj` module, and a project cannot be named `ENTRY`.

[8] In this case, "name" applies to the filename, without its extension

Chapter 4

Operating scenarios

4.1 Developing a native B project

4.1.1 Principle

This section provides an example of a native B project, used to illustrate the following aspects of the development of such a project:

- Influence of the translation into Ada on the architecture of the written B code.
- Unitary translation of B project modules.
- Project linking.
- Compiling and executing the generated code.

The detailed operating information for the Translator will be given by the screen captures of the Atelier B GUI* .

4.1.2 Informal Example Specifications

The purpose is to test the handling of an integer stack. To do this, several modules are written:

- A module that initializes a stack of a given size and which allows to push an element on the stack, or to pop it from the stack.
- A module that displays a stack.

The example will have to create two stacks of different sizes and use the display of their contents to prove that the stacks are correctly initialized and that the push and pop procedures work.

Important remark: This example, and the B source codes presented below for its implementation, do not pretend to be perfect examples of B language. It is not intended as a B language design style example, but simply a complete project creation example, with translation into a target language followed by the execution of the product code.

Thus, to handle unbounded stacks, it would have been possible to implement them on the basic machine `BASIC_ARRAY_VAR` instead of a limited static array.

↻ The examples are not adapted to a HIA translation as the arrays type are not explicitely declared *(refer to appendix B for more details)*. After each example, we will explain the modifications to perform, to translate using the HIA translator.

4.1.3 Project Architecture and B Code

The stack component

The `stack` component models a natural integer stack. The size of the stack can be set by parameter *(it is a machine parameter)*, on condition that it does not exceed 10 elements in order to keep the example simple, the stack is implemented by a static array of 10 integers. The component offers two operations `push` and `pop` that are used to manage the stack.

Figure 4.1 gives the B source code for the specification and the implementation of this component.

↻ To translate into HIA, an explicit array type must be defined :
`table_type = (1..10) --> NAT` and then be used to type `the_stack`.

The display_stack component

The `display_stack` component is used to display a "stack" in the format defined by the `stack` component, i.e, a stack implemented in an array of 10 items. In this way, the `display` operation that is used to display a stack takes as its parameters, not only the array representing the stack, but also the current size of the stack, i.e, the number of array elements that are part of the stack.

For presentation reasons, this operation also takes as its input a message to display before displaying the stack.

Figure 4.2 shows the B source code of the specification and the implementation of this component.

↻ To translate into HIA, an explicit array type must be defined :
`type_array = (1..10) --> NAT` and then, be used to type `stack`. The other components *(for example* `stack`*)* must type their stacks with `type_array` to ensure the translation success.

The stack_interface component

The `stack_interface` component offers a higher level of abstraction than the two preceding components. It is parameterized by the size of the stack to create.

It creates a stack of this size, displays it to ensure that it is empty, fills it with consecutive integers, then displays it to ensure that it is full. The stack will then be emptied of its elements, then displayed to ensure that it is empty.

The stack is created by importing the `stack` component. It is filled in, then emptied using the `push` and `pop` operations in this component. Importing the `display_stack` component

```
MACHINE
   stack(nb_items)

CONSTRAINTS
   nb_items :  NAT &
   nb_items >= 1 &
   nb_items <= 10

CONCRETE_VARIABLES
   the_stack, top_of_stack

INVARIANT
   the_stack :  (1..10)-->NAT &
   top_of_stack :  NAT &
   top_of_stack >= 0 &
   top_of_stack <= nb_items

INITIALISATION
   the_stack ::  (1..10) --> NAT ||
   top_of_stack := 0

OPERATIONS

   push(val) =
   PRE
      top_of_stack < nb_items &
      val :  NAT
   THEN
      the_stack(top_of_stack) := val ||
      top_of_stack := top_of_stack + 1
   END ;

   pop =
   PRE
      top_of_stack >= 1
   THEN
      top_of_stack := top_of_stack - 1
   END

END
```

```
IMPLEMENTATION
   stack_1(nb_items)

REFINES
   stack

INITIALISATION
   the_stack := (1..10)*{0} ;
   top_of_stack := 0

OPERATIONS

   push(val) =
   BEGIN
      top_of_stack := top_of_stack + 1 ;
      the_stack(top_of_stack) := val
   END ;

   pop =
   BEGIN
      top_of_stack := top_of_stack - 1
   END

END
```

Figure 4.1: Specification and implementation of the stack component

```
MACHINE
   display_stack(nb_items)

CONSTRAINTS
   nb_items :  NAT &
   nb_items >= 1

OPERATIONS
   display(
      message,
      pile,
      stack_size) =
   PRE
      message :  STRING &
      pile :  (1..10) --> NAT &
      stack_size :  NAT &
      stack_size <= nb_items
   THEN
      skip
   END
END
```

```
IMPLEMENTATION
   display_stack_1(nb_items)
REFINES
   display_stack
SEES
   BASIC_IO
OPERATIONS
   display(message, stack, stack_size) =
   BEGIN
      STRING_WRITE("Expected result :  ") ;
      STRING_WRITE(message) ;
      STRING_WRITE("\nEffective result :\n") ;
      IF (stack_size = 0)
      THEN
         STRING_WRITE("-- the stack is empty --\n")
      ELSE
         VAR
            ii
         IN
            STRING_WRITE("(bottom of stack) ") ;
            ii := 1 ;
            WHILE ii <= stack_size
            DO
               INT_WRITE(stack(ii)) ;
               STRING_WRITE(" ") ;
               ii := ii + 1
            INVARIANT
               ii :  NAT &
               ii >= 1 &
               ii <= (nb_items + 1)
            VARIANT
               nb_items + 1 - ii
            END ;
            STRING_WRITE("(top of stack)\n")
         END
      END
   END
END
```

Figure 4.2: Specification and implementation of the display_stack component

enables displaying the stack.

Figure 4.3 gives the B source code of the specification and the implementation of this component.

↻ To translate into HIA : the stacks must be typed with the type `type_array` of `display_stack`

The demo component

The `demo` component must trigger the application execution. It has only one operation, that creates and then tests the two stacks.

This component will be the **entry point** of the B project. It is the root of the project import tree, as shown in figure 4.5.

Figure 4.4 gives the B source code for the specification and the implementation of this component.

Among others, this is the one that imports the basic machine `BASIC_IO`[1] as a number of project components will produce displays on-screen *(including the* `demo` *component)* and as previously stated, the rule "an ancestor in the import graph cannot perform `SEES`" must be followed.

4.1.4 Integrating Components into Atelier B

The reader should refer to the [ATB1] document for detailed explanations on the integration of components in Atelier B. This paragraph is limited to a brief description of the steps to follow.

Thus, the user must perform the following operations:

- Create a project. In the rest of this chapter, it will be assumed that the project name is `DEMO_STACKS`. Attach this project to the `LIBRARY` library, therefore it will have access to the basic machine `BASIC_IO`.
- Insert the components detailed in sub-section 4.1.3. Perform the steps detailed in `Type_Check` then in `B0 Check` on these components.

  ↻ For the HIA translator, the B0 Check step must not be performed.
- **Perform the proof of these components**. If this rule is not mandatory in order to perform the translation into target language, it is the only guarantee of the product code quality.

  The translation result of a code not 100% proven is not guaranteed.

4.1.5 Unitary Translation of the Produced Code

Select the four implementations of the project and select `Translator`. In the window displayed, select the target language and `Selected Only`.

[1] Machine that enables implementing inputs/outputs, described in the Appendix A

```
MACHINE
   stack_interface(nb_items)

CONSTRAINTS
   nb_items :  NAT &
   nb_items >= 1

OPERATIONS

   demonstration = skip

END
```

```
IMPLEMENTATION
   stack_interface_1(nb_items)
REFINES
   stack_interface
IMPORTS
   stack(nb_items),
   display_stack(nb_items)
SEES
   BASIC_IO
OPERATIONS
   demonstration =
   BEGIN
      /*?  Display the empty stack ?*/
      STRING_WRITE("Display of the empty stack :") ;
      display("EMPTY STACK", the_stack, top_of_stack) ;
      /*?  Fill the stack ?*/
      STRING_WRITE("Fill the stack and then display it
         :\n") ;
      VAR ii IN
         ii := 1 ;
         WHILE (ii <= nb_items) DO
            push(ii) ;
            ii := ii + 1
         INVARIANT
            ii :  NAT & ii >= 1 &
               ii <= (nb_items + 1)
         VARIANT
            nb_items + 1 - ii
         END
      END ;
      /*?  Display the full stack ?*/
      display("FULL STACK", the_stack, top_of_stack) ;
      /*?  Empty the stack ?*/
      STRING_WRITE("Empty the stack then display it
         :\n") ;
      VAR ii IN
         ii := 1 ;
         WHILE (ii <= nb_items) DO
            pop ;
            ii := ii + 1
         INVARIANT
            ii :  NAT & ii >= 1 &
               ii <= (nb_items + 1)
         VARIANT
            nb_items + 1 - ii
         END
      END ;
      /*?  Display the empty stack ?*/
      display("EMPTY STACK", the_stack, top_of_stack)
   END
END
```

Figure 4.3: Specification and implementation of the interface_stack component

```
MACHINE
   demo

OPERATIONS

   demo_stacks =
      skip

END
```

```
IMPLEMENTATION
   demo_1

REFINES
   demo

IMPORTS
   BASIC_IO,
   interface_A.stack_interface(3),
   interface_B.stack_interface(4)

OPERATIONS

   demo_stacks =
   BEGIN
      STRING_WRITE("-- Beginning of the test\n\n") ;

      STRING_WRITE(
         "-----------------------------------------------\n") ;
      STRING_WRITE(
         "Demonstration with a stack of maximum size 3\n") ;
      STRING_WRITE(
         "-----------------------------------------------\n") ;
      interface_A.demonstration ;

      STRING_WRITE(
         "\n\n-----------------------------------------------\n") ;
      STRING_WRITE(
         "Demonstration with a stack of maximum size 4\n") ;
      STRING_WRITE(
         "-----------------------------------------------\n") ;
      interface_B.demonstration ;

      STRING_WRITE("-- End of the test\n\n")
   END

END
```

Figure 4.4: Specification and implementation of the demo component

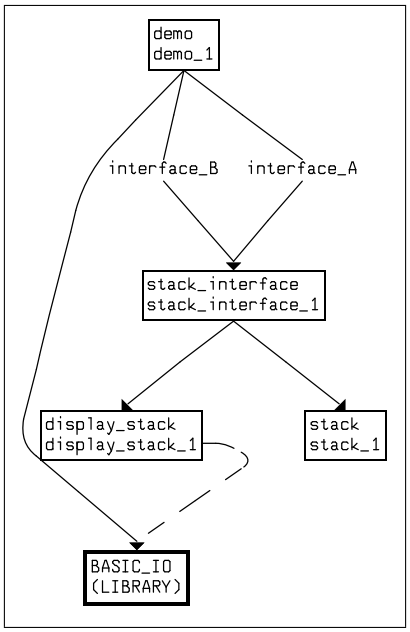

Figure 4.5: Project dependency graph

4.1.6 Project linking

Select implementation `demo_1` and press `Translator`. From the window displayed, select the target language and `All`.

The linker creates a module responsible of the project start-up *(the start-up module)*. When used in the Atelier B GUI* , this module takes the same name as the project.

4.1.7 Compiling and Executing the target code

Open a session on the target machine. In this example, the target system is identical to the development system *(both are Unix systems)*, the target language is Ada and the `gnat` Ada compiler is used.

Move to the directory where the files are translated, and type `make`. Figure 4.6 provides an example of the display obtained.

Warning: The translation of this project with the translator provided with the 3.6 version of Atelier B leads to a translator error.

It is then possible to execute the project. Figure 4.7 presents the result of this execution.

4.2 Developing a Heterogeneous B/Target Language Project

↺ This section does not apply to the HIA translator, that contains some specificities described in paragraph 4.3.

Just like an autonomous project, a heterogeneous project must comprise an entry point. This entry point is a software component(named package in Ada or module in C/C++)

```
/home/ATELIER/B/stack/lang% ls
DEMO_PILES.adb        basic_io.ads        stack_interface.blf        stack.bod
DEMO_PILES.bod        demo.adb            stack_interface.bod        stack.str
display_stack.adb     demo.ads            stack_interface.str        sets.adb
display_stack.ads     demo.blf            makefile                   sets.ads
display_stack.blf     demo.bod            makefile.blf               sets.bod
display_stack.bod     demo.str            stack.adb                  sets.str
display_stack.str     stack_interface.adb stack.ads
basic_io.adb          stack_interface.ads stack.blf
/home/ATELIER/B/stack/lang% make
gcc -I/home/ATELIER/B/LIBRARY/lang -c DEMO_STACKS.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c sets.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c demo.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c basic_io.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c stack_interface.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c stack.adb
gcc -I/home/ATELIER/B/LIBRARY/lang -c display_stack.adb
gnatbl DEMO_STACKS.ali -o DEMO_STACKS
/home/ATELIER/B/stack/lang% ls -l DEMO_STACKS
-rwxr-xr-x 1 ATELIER bin 205899 May 17 11:35 DEMO_STACKS*
```

Figure 4.6: Compilation of an Ada code produced by the Ada translator

allowing the creation of the physical instances of the components and their correct initialization. It then allows access to all these instances.

This package exports an `INITIALISATION` procedure and a, depending on the target language :

- In Ada, a `this` object.
- In C++, an `entry` object.
- In C, an `entry` structure.

The `INITIALISATION` procedure creates the physical instances, and `this/entry` is used to reference project objects. Thus, `this/entry` references the instance of the project entry point *(i.e, the instance of* `demo` *in this example)*, and then it is possible to recursively scan the dependency graph of the project by applying the following rule:

- If instance I of the graph is reached thanks to path Λ.
- If I uses[2] an instance J
- Then instance J is reached by path Λ`.ref_`J in Ada or Λ`->ref_`J in C/C++.

In this way, by referring to the dependency graph illustrated by figure 4.5 on page 26, it is possible to build a table that assigns to each physical instance of the project an "access path". Figure 4.8 presents this table for our example.

[2]In the IMPORTS, SEES or EXTENDS meaning

```
-- START OF TEST

---------------------------------------------
Demonstration with a stack of maximum size 3
---------------------------------------------
Display the empty stack:  Expected result:  STACK EMPTY
Effective result:
-- the stack is empty --
The stack is filled and displayed:
Expected result:  STACK FULL
Effective result:
(bottom of stack) 1 2 3 (top of stack)
The stack is emptied and displayed:
Expected result:  STACK EMPTY
Effective result:
-- the stack is empty --


---------------------------------------------
Demonstration with a stack of maximum size 4
---------------------------------------------
Display the empty stack:  Expected result:  STACK EMPTY
Effective result:
-- the stack is empty --
The stack is filled and displayed:
Expected result:  STACK FULL
Effective result:
(bottom of the stack) 1 2 3 4 (top of the stack)
The stack is emptied and displayed:
Expected result:  STACK EMPTY
Effective result:
-- the stack is empty --
-- END OF TEST
```

Figure 4.7: Executing the Ada program

Instance	Ada acces path	C,C++ acces path
demo	this	entry
BASIC_IO	this.ref_BASIC_IO	entry→ref_BASIC_IO
interface_A.stack_interface	this.ref_interface_A	entry→ref_interface_A
interface_B.stack_interface	this.ref_interface_B	entry→ref_interface_B
stack created by interface_A.stack_interface	this.ref_interface_A.ref_stack	entry→ref_interface_A →ref_stack
stack created by interface_B.stack_interface	this.ref_interface_B.ref_stack	entry→ref_interface_B →ref_stack
display_stack created by interface_A.stack_interface	this.ref_interface_A.ref_stack	entry→ref_interface_A →ref_display_stack
display_stack created by interface_B.stack_interface	this.ref_interface_B.ref_stack	entry→ref_interface_B →ref_display_stack

Figure 4.8: Correspondence between the physical instances of the project and their access path in Ada, C and C++

It is then possible to call the component operations, without forgetting to send them as the first argument, the implicit instance parameter, i.e, the pointer that allows them to be reached. In this way, it is possible to call the component method `display` for component `display_stack` created by `interface_A` by writing the following code:

Language	Code
Ada	`display_stack.display(this.ref_interface_A.ref_display_stack, ...) ;`
C	`display_stack->display(entry->ref_interface_A->ref_display_stack, ...) ;`
C++	`entry->interface_A->display_stack->display( ...)`

4.3 Developement of a Heterogeneous B/HIA project

↺ This section applies to the HIA translator only. The development of heterogeneous projects for the other translators is described in 4.2

The code generated by the HIA translator is quite simple and close from the B0. A package is created for each instance of a component. Each package has the same name as the associated component, prefixed by the rename component prefix, followed by a '_' character.

The constants, the formal parameters and the components variables are translated in the package. The operations have the same name and signature as in B.

Therefore, the development of a heterogeneous project B/HIA does not require additional work compared to a Ada "HIA" native development.

Chapter 5

Complete List of Services

5.1 Unitary Translation of an Implementation

trad `[OPTIONS] -i nom_implementation[.extension]`, where `OPTIONS` is a combination of the options presented in table 5.2 and *trad* is given table 5.1.

Translator	**Software**
Ada	tbada
HIA	tbhia
C++	tbcpp
C	tbcpp -c

Figure 5.1: Translation of an implementation from the command line

5.2 Project Linking

trad `[OPTIONS] -o executable_name -e spec_input_point -E bed_path`
where `OPTIONS` is a combination of the options presented in table 5.3 and *trad* is given by table 5.1.

Remark: An option is used to change the name of the target compiler when performing linking. The produced `makefile` file instances the variable `ADA_COMPILE` (or `CPP_COMPILE`) with this value. It is still possible to change this value afterwards, either by modifying it directly in the makefile, or by passing it in the command line as shown in the example below:

```
make ADA_COMPILE=my_ada_compiler
make CPP_COMPILE=my_c++_compiler
```

5.3 Translating a Project with Traces

↺ This section applies to the Ada translator only

Option	Semantics	Default value Ada/HIA Translators	Default value C/C++ Translator
B	Change the suffix of object files of type "package body"	`bod`	`bdy`
D	Prints configuration before translating		
C	Software compilation information		
I	Add a path to search for B source files		
l	Change the maximum length of producted lines	`80`	`80`
O	Change the suffix of object files of type "B Link File"	`blf`	`blf`
P	Change the output path	`../lang`	`../lang`
S	Change the suffix of object files of type "package specification"	`str`	`spe`
t	Change the value in characters of a tabulation	`4`	`4`
T	Ada : generate code whose execution yields traces about called operations and parameter values		
v	Verbose mode		
V	Software version number and usage		
w	Adds an "unproved component" message to generated code *(used automatically by Atelier B)*		

Figure 5.2: Options available when performing the unitary translation of a component

Semantic	tbada parameter Default value	tbcpp parameter Default value
Change the target compiler name	A, `gnatgcc`	T, `gcc`
Change the suffix of object files of type "package body" of type "package body"	B, `bod`	B, `bdy`
Print names clashes with their resolution	c	c
Software compilation information	C	I
Print configuration before linking	D	D
Change the name of target linker	K, `gnatbl`	K, `gcc`
Change the maximum character length of a code line	l, `80`	l, `80`
Add search path for library object files	L	L
Change `MAXINT` value	M, `2147483647`	M, `2147483647`
Change `MININT`value	N, `-2147483647`	m, `-2147483647`
Linking for an heterogenous project	n	n
Change the suffix for object files of type "B Link File"	O, `blf`	O, `blf`
Change the directory where files are created	P, `../lang`	P, `../lang`
Change the suffix for object files of type "package specifications"	S, `str`	S, `spe`
Change the value in characters of a tabulation	t, `4`	t, `4`
Create Ada code with traces about called operations and parameter values	T	non available
Verbose mode	v	v
Version number and software usage	V	V

Figure 5.3: Options available during linking

The `-T` option of the Ada translator is used to perform a project translation with traces. This option allows:

- In a unitary translation, to request a module translation with the production of traces.
- At linking level, to request the addition of trace modules when compiling is performed.

A project that has been translated with traces, will produce a complete trace of the function calls when it is executed. This trace is produced in the `.trace` file, located in the directory where the project is started. This file, identifies:

- Function calls, that display the implicit instance as well as the value of input parameters.
- Function outputs, with a display of the implicit instance as well as the value of output parameters.

Thus, a translation with traces allows the user to monitor the progress of their project precisely, without needing to add instructions for this purpose in the B source code.

5.4 Restriction of translators use

Warning: The ADA, C and C++ translators provided in version 3.6 of Atelier B are experimental. Their goal is to show that it is possible to translate B0 implementations into classical programming languages. Their use is then not guaranteed.

The aim of this section is to define all the translator known anomalies.

- Package name:

  The ADA translators accept to generate packages having the same name as the predefined packages of the language. These packages cannot be compiled. To get around this anomaly, give to the B components a different name from the predefined ADA packages ones.

- Translation of the arrays:

  The arrays translation into ADA using the reusable components (machines of the L_ARRAY family) may be incorrect. The ADA translator can allocate a big space for the table (integer) and then overload the memory.

- Comparison between record fields of enumerated type:

  When comparing ('=' or '/=') two enumerated or record tables, which are themselves record field coming from an external machine, the translators wrongly translate: rec'field = enum instead of: mch."="(rec'field, enum).

- Set formal parameters:

  The translation of set formal parameters using the translators provided with Atelier B is sometimes incorrect. It is not advised to use the same name for different set formal parameters.

- Use of the reusable components:

  The reusable components provided with Atelier B use set formal parameters. As a consequence of the previously mentionned anomaly, their use with the ADA, C and C++ translators may lead to translator error or to a false target code that cannot then be executed.

Chapter 6

Glossary

API Application Program Interface. The external interface provided by a system or a library.

Batch Mode An Atelier B command-line operating mode executing automatic procedures described in the form of command files.

GUI Graphical User Interface. This interface is graphical, the interface in text by command line mode is the batch mode.

Offensive code An offensive code is a code which assumes that the user obeys a "contract", i.e, that certain conditions are met. In this way, these conditions are not tested in the code. In C, the `strcpy` and `strcmp` type functions are examples of an offensive code as they do not verify the integrity of their arguments *(memory fields allocated by the user, strings ending with* `\0`*, ...)*.

SCCS *(Source Code Control System)*. A set of tools that allows the management of source files and, by extension, the management of binary files resulting from compiling these source files.

Shell An interface program between the user and the operating system. In Unix, the main shells are `sh`, `ksh`, `bash` and `csh`.

Appendix A

Basic Machines

A.1 Principle

A.1.1 Definition

A basic machine is a machine with a specification in B which is directly implemented in the target language.

A.1.2 Usefulness

Basic machines are used to implement functions that cannot be performed in B0. Most of the time, these functions are the ones that are close to system level: input/output, dynamic memory management, ...

Atelier B is supplied as standard with a collection of basic machines that allow the creation of B projects that interact with an operator and which use complex data structures. Sub-section A.2 describes these machines. However, an Atelier B user may still need to implement his own basic machines. Sub-section A.3 describes how to perform this task.

A.2 Description of the Basic Machines Delivered with Atelier B

Table A.1 provides the list of the basic machines supplied with Atelier B. Manual [ATB2] provides a complete description of these machines: the reader who wishes to find more information about the use of these machines may refer to it.

Some specification functionnalities can only be implemented through basic machines. For instance, the only way to implement a "dynamic" array [1] is to import `BASIC_ARRAY_VAR` (one dimension)*(or* `BASIC_ARRAY_RGE` *two dimensions).*

A.3 Writing a Basic Machine

A basic machine written by the user comprises four elements:

[1] i.e. a table whose size is machine parameter dependent.

MACHINE	DESCRIPTION
`BASIC_ARRAY_RGE`	Implementing a two dimensional table
`BASIC_ARRAY_VAR`	Implementing a one dimensional table
`BASIC_IO`	Basic Inputs/Outputs

Figure A.1: Basic machines supplied with Atelier B

1. A specification in B.
2. An interface (or specification), written in target language
3. A "package body", written in target language
4. A file in "B Link File" format destined for the linker.

The easiest way to write a basic machine is as follows :

1. Write the specification in B.
2. Write an empty shell implementation, i.e, where the body of all operations contains `skip`.
3. Use Atelier B to translate the implementation.
4. Keep the B link file, and fill the obtained skeletons with the desired code.

A.3.1 The B Specification writing method

The B specification is written according to the usual rules that apply when writing a component in B language. It is interesting to write a specification that describes the effect of the basic machine's operations as precisely as possible, rather that limiting the specification to empty shells *(i.e,* `skip`*)* for the specification. This way, the proof mechanism guarantees the author of the basic machine's Ada code that a certain number of constraints are met, making it possible to write offensive* code.

A.4 Target code (interface and body) writing method

Target code must comprise a package specification and the package body, located in two files. The respective default extensions of these two files are given in table 5.2. If the Translator options changing these extensions[2] are used, it is important to ensure that this change is also made to the extensions of files that make up the basic machines. Paragraphs A.4.1, A.4.2, A.4.3 and A.4.4 describe target code writing methods for the Ada, HIA, C++ and C translators.

[2]These options are described in Chapter 5.

Type B0	Type Ada/HIA	Type C/C++
INT	INTEGER	T_int
NAT	INTEGER	T_nat
NAT1	INTEGER	T_nat
BOOL	BOOLEAN	T_bool
STRING	STRING	T_string
formal parameter of set type	↻ generic package item (ADA) ↻ Sub-type (HIA)	T_set *
array	array	T_array_x * x = array. dim

Figure A.2: Translation of the B0 types in Ada, HIA, C and C++

A.4.1 Target Code writing method for Ada

↻ This section applies to the Ada translator only

- The name of the package is the same as the name of the B specification, and it is not case sensitive (uses upper or lower case letters). However, the filenames used must be in lower case.

  Let P be this name.

- Package P must define:

  - Type TYPE_P which is a `record` that must comprises an `initialisation` field of BOOLEAN type. In this `record`, will be placed all the data structures required for modeling instances of each machine.
  - Type PTR_P, is a "pointer to P" type.
  - An IMPORTS procedure which takes as input the `this` parameter that is a PTR_P type and which describes the instance to import, followed by as many input parameters as there are scalar type formal parameters. Figure A.2 shows the correspondence between B0 and Ada types.
  - An INITIALISATION procedure which takes as input a `this` parameter that is a PTR_P type and which describes the initialisation of the instance.
  - As many procedures as there are machine operations. The procedures take the same name as in the B specification, preceded by the name of the machine in lower case, framed by the # characters *(i.e, in* `#machine#operation` *format)*. Their parameters are obtained as follows:
    - The first parameter, called the implicit instance, of type TYPE_P and points to the component instance where the operation will be performed.
    - Then come the operation's input parameters, in the order of their declaration in B. These are `in` mode parameters. The types are obtained as shown in the table in figure A.2.
    - Then come the operation's output parameters, in the order of their declaration in B. These are `in out` mode parameters. The types are obtained as shown in the table in figure A.2.

A.4.2 Target code writing method for HIA

↻ This section applies to the HIA translator only

- The package name is the same as the name in B specification, using indifferently upper or lower cases. However, the filenames used must be in lower cases. Let P be this name.
- Package P must define :
  - A clause `with` for each required machine.
  - The translation of the `INITIALISATION` procedure, located in the initialisation function of the package.
  - As many procedures as there are machine operations. The procedures have the same name as in the B specification, preceded by the machine name in lower cases, framed by the `#` characters *(i.e, in* `#machine#operation` *format).* Their parameters are obtained as follows :
    - The operation's input parameters, in the order of their declaration in B. These are parameters of `in` mode. The types are obtained as shown in the table in figure A.2.
    - The operation's output parameters, in the order of their declaration in B. These are parameters of `in out` mode. The types are obtained as shown in the table in figure A.2.

A.4.3 Target code writing method for C++

↻ This section applies to the C++ translator only

- The name of the class is the same as the name of the B specification, prefixed by `T_`
  Let P be this name.
- Class `T_P` must define:
  - Class `TYPE_`P which must contain an `initialisation` field of type `T_bool`. In this class will be placed all of the data structures required for modeling instances of each machine.
  - An `IMPORTS` member function which describes the instance to import, followed by as many input parameters as there are scalar type formal parameters. Figure A.2 shows the correspondence between B0 and C++ types.
  - An `INITIALISATION` member function which describes the initialisation of the instance.
  - As many member functions as there are machine operations. The functions have the same name as in the B specification, preceded by the name of the machine in lower case, framed by the `#` characters *(i.e, in* `#machine#operation` *format).* Their parameters are obtained as follows:

* The operation's input parameters, in the order of their declaration in B. These are `in` mode parameters. The types[3] are obtained as shown in the table in figure A.2.
* Then come the operation's output parameters, in the order of their declaration in B. These are `in out` mode parameters. The types are obtained as shown in the table in figure A.2.

A.4.4 Target code writing method for C

↻ This section applies on the C translator only

- The name of the pseudo-class is the same as the name of the B specification, prefixed by `T_`

  Let P be this name.

  (The pseudo-class is a structure which is only accessed through functions, prefixed by the class name, and which emulate the methods of a real class.)

- Pseudo-class `T_P` must define :
  - An `initialisation` field of type `T_bool`.
  - A field for each data structures required for modeling instances of each machine *(formal parameter, concrete variable, ...)*. Field names are prefixed by the name of the machine, framed by `#` characters.

- Methods of the pseudo-class `T_P` are emulated by the following functions :

- A `new_T_P` function+. It is the equivalent of the class constructor. It takes as input a parameter `_this` of type `T_p *`.

- A `T_P_IMPORTS` function which takes as input a parameter `_this` of type `T_P *`, followed by as many input parameters as there are scalar or set type formal parameters. Figure A.2 shows the correspondence between B0 and C types.

- A `T_P_INITIALISATION` function which takes as input a parameter `_this` of type `T_P *` and which describes the initialisation of the instance.

- As many functions as there are machine operations. The functions have the same name as in the B specification, preceded by the name of the machine in lower case, framed by the `#` characters, and the pseudo-class name *(that is to say, in the* `#machine#T_P_operation` *format)*. Their parameters are obtained as follows:
  - The first parametert is `_this` of type `T_P *`.
  - The operation's input parameters, in the order of their declaration in B. These are `in` mode parameters. The types[4] are obtained as shown in the table in figure A.2.
  - Then come the operation's output parameters, in the order of their declaration in B. These are `in out` mode parameters. The types are obtained as shown in the table in figure A.2.

[3] Classes T_set and T_array are defined in predefined components installed with the translator
[4] Classes T_set and T_array are defined in builtin components installed with the translator

```
--************************************************************************ *****
--
--Basic machine BASIC_IO for ADA target language
--
--(C) 1996 STERIA
--
--Version @(#) basic_io.blf 1.2 (date :  22 Mar 1996)
--
--************************************************************************ *****
_BEGIN_CLASS
   -- Machine name
   BASIC_IO
   _BEGIN_LEVEL_1 -- Class level
      -- operations
      INTERVAL_READ
      INT_WRITE
      BOOL_READ
      BOOL_WRITE
      CHAR_READ
      CHAR_WRITE
      STRING_WRITE
   _END_LEVEL_1
_END_CLASS
```

Figure A.3: Example of a "B Link File"

A.4.5 The "B Link File"

The "B link File" is made of several sections delimited by `BEGIN_SECTION_NAME` and `END_SECTION_NAME` keywords. These sections can be empty or even not be present. They contain data used to manage name conflicts that occur when linking various modules as well as the conflicts with the target language. They also enable to identify the array and record types and the constants exported by the module.

If the code of the basic machine is correct and uses valid identifiers for the chosen target language, the user only has to give the machine name *(*`CLASS` *section)*, and to fill in the section `LEVEL_1` of `CLASS` with the names of the operations exported by the machine.

This way, the name of the operations exported by the machine will never clash with the other modules names.

↺ Beware that for the Ada and HIA translators, and contrary to Ada language conventions, operations names are case sensitive, i.e, the usage of upper and lower case letters shown in the basic machine B language source must be followed.

Figure A.3 provides an example of the "B Link File" provided for the basic machine `BASIC_IO`.

Appendix B

B0 specifities accepted by the HIA translator

B.1 Introduction

The HIA translator uses its own B0 language. These specifities were introduced:

- On the one hand, because of the explicit typing per arrays and records identifier,
- On the other hand, because we want to generate simple code, and especially, we want the Ada packages to be close to the B components by gathering the constants and formal parameters declarations.

We will name this language HIA-B0 in the rest of this chapter.

B.2 Arrays translation

B.2.1 Principle

To be translated by the HIA translator, an array must have been explicitly typed with an identifier *(this is a restriction compared to classical B0).*

Such an identifier is a concrete constant that has been typed as equal to a B "array" type.

This B type defined in the clause `PROPERTIES` is used for the translation; the valuation is ignored.

It is advised to copy the typing of the clause `PROPERTIES` in the clause `VALUES` to avoid any confusion. The possibility to define a concrete constant of array type is an extension according to the classical B0 language.

B.2.2 Example

To write in HIA-B0 the declaration of a var variable of type " integers array, which index are between 4 and 12 ", we should write :

```
CONCRETE_CONSTANTS
 array_type

PROPERTIES
 array_type = (4..12) --> INT  / * B0-HIA extension * /

VALUES
 array_type = (4..12) --> INT / * unused by the translator  * /

CONCRETE_VARIABLES
 var

INVARIANT
 var : array_type / * explicit typing with an identifier as ruled by HIA-B0 * /

INITIALISATION
     var := (4..12)*{1}
```

B.3 Record translation

B.3.1 Principle

To be translated by the HIA translator, a record must have been explicitly typed with an identifier *(this is a restriction compared to classical B0)*.

Such an identifier is a concrete constant that has been typed as equal to a B "record" type.

This B type defined in the clause `PROPERTIES` is used for the translation; the valuation is ignored.

It is advised to copy the typing of the clause `PROPERTIES` in the clause `VALUES` to avoid any confusion. The possibility to define a concrete constant of array type is an extension according to the classical B0 language.

B.3.2 Example

To write in HIA-B0 the declaration of a var variable of type "record", which the first tab field is of type "integer array", which index are between 4 and 12, and the second valid field is of 'boolean' ", we should write:

```
CONCRETE_CONSTANTS
 array_type,
 record_type

PROPERTIES
 array_type = (4..12) --> INT  / *  HIA-B0 extension * / &
 record_type = struct(
  tab : array_type,
  valid : BOOL) / * HIA-B0 extension * /

VALUES
 array_type = (4..12) --> INT ; / * unused by the translator * /
 record_type = struct(
  tab : array_type,
```

```
  valid : BOOL) / * unused by the translator * /

CONCRETE_VARIABLES
 var

INVARIANT
 var : record_type / * explicit typing with an identifier as ruled by HIA-B0 * /

INITIALISATION
     var := rec( (4..12)*{1}, FALSE)
```

B.4 Formal parameters

The effective formal parameters are directly translated in the package of the concerned instance. For example, if `Mch` is a B component having a scalar formal parameter `param` and if the project's two instances `i1.Mch(5)` and `i2.Mch(10)` of`mch` are created, so:

- in the package `i1_Mch` a constant `param` with value `5` is defined
- in the package `i2_Mch` a constant `param` with value `10` is defined

The effective formal parameters are then declared in the associated packages. As a consequence, they have the scope of these packages.

Now, let us assume that a component `Mch1` uses a component `Mch2(param2)`, and that the valuation of `param2` involves the data of `Mch1`, for example a constant `cst1` of `mch1`:

`param2 = Mch1.cst1`

The value of `Mch2.param2` then involves `Mch1.cst1`. But `Mch1.cst1` is not visible from the package `Mch2` *(as* `Mch1` *uses* `Mch2`*, then the package* `Mch1` *makes a* `with` *of* `Mch2` *:* `Mch2` *is then unable to make a* `with Mch1` *without risking a cyclical dependancy).*

It is not possible to compile the obtained code. More generally, we can see that it is not possible to use non literal data for the valuation of the machine formal parameters.

Consequence : The effective formal parameters of the machines allowed in HIA-B0 are the literals (integers or booleans).

Bibliography

[ANSI-C++] Le langage C++, troisième édition
Bjarne STROUSTRUP
Campus Press
ISBN 2-7740-0609-2

[CPP1] The C++ language, Addison-Wesley
Bjarne Stroustrup

[ANSI-C] The C language, ANSI normalisation
B.W. Kernighan et D.M. Ritchie

[ADA-83] Manuel de référence du langage de programmation Ada
Alsys
Février 1987

[ADA-95] Ada 95
Deuxième édition
John BARNES
Addisson-Wesley
ISBN 0-201-34293-6

[ADA1] Manuel de référence du langage de programmation Ada
Alsys
Février 1987

[ADA2] Ada, An Introduction. Ada reference manual.
Henry Ledgard
Springer-Verlag
ISBN 0-387-90568-5 et 3-540-90568-5

[ADA3] Ada programming manual
Integrated computer systems
NEW/8/80

[SPARK] High integrity Ada
The SPARK approach
John BARNES
Addisson-Wesley
ISBN 0-201-17517-7

[ATB1] Atelier B reference manual

[ATB2] Reusable components. Reference manual.

Index

Ada package
    body, 8
    specifications, 8
array
    translation, 7
Atelier B
    Batch mode, 15
    GUI, 15
Atelier B GUI, 11
autonomous, 19

B0
    checker, 5
    implementation, 5
basic machine, 9, 37
    offensive code, 38
    supplied with Atelier B, 37
    written by the user, 37
Batch
    Atelier B mode, 12
Batch mode, 15
Batch mode of Atelier B, 12

C component
    body, 8
    specifications, 8
C++ class
    body, 8
    specifications, 8
compilation, 5
    project makefile, 9
compiling, 14
cross-translation, 5

entry point, 9, 23, 27
environment, 5
    HP-UX, 5
    Linux, 5
    Solaris, 5
    Sun Os, 5

GUI, 15

hétérogène, 29
heterogeneous, 26
HP-UX, 5

identifier clash, 5, 6
implicit instance, 29, 39

library, 6
limits checker, 17
linking, 8, 9, 13, 31
Linux, 5

makefile, 9

project
    entry point, 9, 23, 27
    start-up module, 8, 9, 26
Proof, 16

Restrictions on the names of modules and projects, 17

sets, 9
Solaris, 5
start-up module, 8, 9, 26
Sun Os, 5

Translation
    and proof, 16
translation, 5, 8, 11, 12, 31
    entry point, 9, 27
    implicitly glued data, 7
    MAXINT, 16
    MININT, 16
    renaming, 7
    size of lines, 16
    start-up module, 8, 9
    with trace, 31